Ownership and Borrowing in Rust

Michael Birch

University of Waterloo, May 2023

GitHub Repository

- Code examples found here

Why Rust?

- Garbage collection creates overhead
  - E.g., Java
- Manual memory management can cause bugs
  - Memory leaks
  - Invalid memory access (seg fault)
- Rust's solution: compile-time "borrow checker"
  - Bonus: also prevents data races

Drop

- Prevents memory leaks
- Enables RAII (Resource Allocation is Initialization)

Drop

```rust
#[test]
▶ Run Test | Debug
fn test_drop() {
    let _alice: CustomDrop = CustomDrop { name: "Alice" };
    let x: u32 = {
        let _bob: CustomDrop = CustomDrop { name: "Bob" };
        42
    };
    println!("{x}");
}

#[derive(Debug)]
2 implementations
pub struct CustomDrop {
    pub name: &'static str,
}

impl Drop for CustomDrop {
    fn drop(&mut self) {
        println!("Dropping {}", self.name);
    }
}
```

```
cargo test test_drop -- --nocapture

running 1 test
Dropping Bob
42
Dropping Alice
test drop::test_drop ... ok
```

Move

- Variables can be **moved** into another scope or data type

```rust
#[test]
▶ Run Test | Debug
fn test_move() {
    let _container: Owner<CustomDrop> = {
        let alice: CustomDrop = CustomDrop { name: "Alice" };
        Owner { inner: alice }
    };
    println!("End of scope");
}
```

```
cargo test test_move -- --nocapture

running 1 test
End of scope
Dropping owner container
Dropping Alice
test ownership::test_move ... ok
```

Move

- Variables can be **moved** into another scope or data type

```rust
#[test]
▶ Run Test | Debug
fn test_function_move() {
    fn move_var<T: std::fmt::Debug>(value: T) {
        println!("Taken ownership of {value:?}");
    }

    let alice: CustomDrop = CustomDrop { name: "Alice" };
    move_var(alice);
    println!("End of scope");
}
```

```
cargo test test_function_move -- --nocapture

Taken ownership of CustomDrop { name: "Alice" }
Dropping Alice
End of scope
test ownership::test_function_move ... ok
```

Move

- Variables can be **moved** into another scope or data type
- A variable must have a single owner to prevent invalid memory access (seg fault)

```rust
fn test_double_move() {
    let alice: CustomDrop = crate::drop::CustomDrop { name: "Alice" };
    let _owner1: Owner<CustomDrop> = crate::ownership::Owner {inner: alice};
    let _owner2: Owner<CustomDrop> = crate::ownership::Owner {inner: alice};
}
```

```
   Compiling borrowing-and-ownership v0.1.0 (/mnt/c/Users/mbirc/Documents/rust/borrowing_and_ownership)
error[E0382]: use of moved value: `alice`
  --> src/ownership.rs:51:33
   |
49 |     let alice = crate::drop::CustomDrop { name: "Alice" };
   |         ----- move occurs because `alice` has type `CustomDrop`, which does not implement the `Copy` trait
50 |     let _owner1 = Owner {inner: alice};
   |                                 ----- value moved here
51 |     let _owner2 = Owner {inner: alice};
   |                                 ^^^^^ value used here after move
```

Copy

- Tells the compiler to copy a value if it is moved multiple times
- Automatically implemented for primitive types like `u32`

```rust
#[test]
▶ Run Test | Debug
fn test_copy() {
    let alice: Copyable = Copyable { name: "Alice" };
    let _owner1: Owner<Copyable> = crate::ownership::Owner { inner: alice };
    let _owner2: Owner<Copyable> = crate::ownership::Owner { inner: alice };
    println!("End of scope");
}
```

Borrowing

- Borrowing allows accessing a value without moving or copying it
- Borrowed values are called “references”

```rust
#[test]
▶ Run Test | Debug
fn test_borrow() {
    fn borrow_to_print<T: std::fmt::Debug>(value: &T) {
        println!("Borrowed {value:?}");
    }

    let alice: CustomDrop = crate::drop::CustomDrop { name: "Alice" };
    borrow_to_print(&alice);
    borrow_to_print(&alice);
    println!("End of scope");
}
```

```
cargo test test_borrow -- --nocapture

running 1 test
Borrowed CustomDrop { name: "Alice" }
Borrowed CustomDrop { name: "Alice" }
End of scope
Dropping Alice
test borrow::test_borrow ... ok
```

Borrowing

- Data types can hold references, but require a “lifetime” parameter

```
0 implementations
pub struct Borrower<'a, T> {
    pub value: &'a T,
}

#[test]
▶ Run Test | Debug
fn test_data_type_borrow() {
    let alice: CustomDrop = crate::drop::CustomDrop { name: "Alice" };
    let _x: u32 = {
        let _borrowed1: Borrower<CustomDrop> = Borrower { value: &alice };
        let _borrowed2: Borrower<CustomDrop> = Borrower { value: &alice };
        42
    };
    println!("End of scope");
}
```

Borrowing

- Data types can hold references, but require a “lifetime” parameter
- Lifetimes are how the compiler keeps track of when values are dropped to prevent seg faults

```rust
fn test_value_does_not_live_long_enough() {
    let _borrow = {
        let alice = crate::drop::CustomDrop { name: "Alice" };
        Borrower { value: &alice }
    };
}
```

```
error[E0597]: `alice` does not live long enough
  --> src/borrow.rs:52:27
   |
50 |     let _borrow = {
   |         ------- borrow later stored here
51 |         let alice = crate::drop::CustomDrop { name: "Alice" };
   |             ----- binding `alice` declared here
52 |         Borrower { value: &alice }
   |                           ^^^^^^ borrowed value does not live long enough
53 |     };
   |     - `alice` dropped here while still borrowed
```

Cloning

- `Clone` is similar to `Copy`, but the compiler does not automatically make the copy
- `Clone` operations are supposed to be much more expensive than `Copy`
  - (e.g. require heap allocation)
- `Clone` operates on a reference

```
pub trait Clone {
    fn clone(&self) -> Self;
}
```

Cloning

- `Clone` is similar to `Copy`, but the compiler does not automatically make the copy
- `Clone` operations are supposed to be much more expensive than `Copy`
  - (e.g. require heap allocation)
- `Clone` operates on a reference

```rust
#[test]
▶ Run Test | Debug
fn test_clone() {
    let alice: Clonable = Clonable {
        name: "Alice".into(),
    };
    let _owner1: Owner<Clonable> = crate::ownership::Owner {
        inner: alice.clone(),
    };
    let _owner2: Owner<Clonable> = crate::ownership::Owner { inner: alice };
    println!("End of scope");
}
```

Reference Counting

- Rc (and Arc) is a special type in the standard library to “cheat” the ownership rule
- The object is owned by an allocation on the heap; it is dropped when the last reference object is dropped.
- Cloning references does not duplicate the underlying object

Reference Counting

```rust
#[test]
▶ Run Test | Debug
fn test_reference_counted() {
    use std::rc::Rc;

    let alice: CustomDrop = crate::drop::CustomDrop {name: "Alice"};
    let rc1: Rc<CustomDrop> = Rc::new(alice);
    let rc2: Rc<CustomDrop> = Rc::clone(self: &rc1);
    let rc3: Rc<CustomDrop> = Rc::clone(self: &rc1);
    drop(rc1);
    println!("Dropped one reference");
    drop(rc2);
    println!("Dropped second reference");
    drop(rc3);
    println!("Dropped final reference");
}
```

```
cargo test test_reference_counted -- --nocapture

running 1 test
Dropped one reference
Dropped second reference
Dropping Alice
Dropped final reference
test reference_counted::test_reference_counted ... ok
```

Mutable Borrowing

- Mutable references are distinct from immutable ones
- Any number of immutable references are allowed; only one mutable is allowed
- This prevents data races

```
#[test]
▶ Run Test | Debug
fn test_mut_borrow() {
    fn transmute_to_bob(value: &mut CustomDrop) {
        value.name = "Bob";
    }

    let mut alice: CustomDrop = CustomDrop {name: "Alice"};
    transmute_to_bob(&mut alice);
    println!("End of scope");
}
```

```
cargo test test_mut_borrow -- --nocapture

running 1 test
End of scope
Dropping Bob
test borrow::test_mut_borrow ... ok
```

Mutable Borrowing

- Mutable references are distinct from immutable ones
- Any number of immutable references are allowed; only one mutable is allowed
- This prevents data races

```
#[test]
▶ Run Test | Debug
fn test_mut_borrow() {
    fn transmute_to_bob(value: &mut CustomDrop) {
        value.name = "Bob";
    }

    let mut alice: CustomDrop = CustomDrop {name: "Alice"};
    let reference: &CustomDrop = &alice;
    transmute_to_bob(&mut alice);
    println!("{reference:?}");
    println!("End of scope");
}
```

```
error[E0502]: cannot borrow `alice` as mutable because it is also borrowed as immutable
  --> src/borrow.rs:61:22
   |
60 |     let reference = &alice;
   |                     ------ immutable borrow occurs here
61 |     transmute_to_bob(&mut alice);
   |                      ^^^^^^^^^^ mutable borrow occurs here
62 |     println!("{reference:?}");
   |              --------- immutable borrow later used here
```

Interior mutability

- Allows “cheating” the Rust type system by mutating an object via an immutable reference
- Sometimes necessary in complex concurrent applications

```rust
fn test_data_race() {
    use std::{thread, time::Duration};

    // Define a cell with value equal to 40
    let cell: SimpleCell = SimpleCell::new(40);
    let borrow: &SimpleCell = &cell;
    // Define a function which will increment the value of the cell by 1.
    let add_one: impl Fn() = || {
        let mut x: u32 = borrow.get();
        x += 1;
        thread::sleep(dur: Duration::from_millis(10));
        borrow.set(new_value: x);
    };
    std::thread::scope(|s: &Scope| {
        // Create two threads, both incrementing the cell by 1.
        let thread1: ScopedJoinHandle<()> = s.spawn(add_one);
        let thread2: ScopedJoinHandle<()> = s.spawn(add_one);

        thread1.join().unwrap();
        thread2.join().unwrap();
    });

    assert_eq!(cell.get(), 42);
}
```

- Using unsafe code we can write an example of a data race

```
thread 'interior_mut::test_data_race' panicked at
'assertion failed: `(left == right)`
  left: `41`,
 right: `42`', src/interior_mut.rs:34:5
```

- Using types in the standard library, interior mutability can be done safely

```rust
fn test_safe_interior_mutability() {
    use std::{sync::RwLock, thread, time::Duration};

    let cell: RwLock<u32> = RwLock::new(40_u32);
    let borrow: &RwLock<u32> = &cell;
    let add_one: impl Fn() = || {
        let mut x: RwLockWriteGuard<u32> = borrow.write().unwrap();
        thread::sleep(dur: Duration::from_millis(10));
        *x += 1;
    };
    std::thread::scope(|s: &Scope| {
        // Create two threads, both incrementing the cell by 1.
        let thread1: ScopedJoinHandle<()> = s.spawn(add_one);
        let thread2: ScopedJoinHandle<()> = s.spawn(add_one);

        thread1.join().unwrap();
        thread2.join().unwrap();
    });
    assert_eq!(*cell.read().unwrap(), 42);
}
```

```
cargo test test_safe_interior_mutability -- --nocapture

running 1 test
test interior_mut::test_safe_interior_mutability ... ok
```

Exercise: Map-Reduce

```rust
pub fn map_red<T, U, M, R>(data: &[T], map_fn: M, reduce: R, init: U) -> U
where
    // Don't worry about the Send + Sync everywhere, that's just to tell rust the data
    // can be shared between threads safely.
    T: Send + Sync,
    U: Clone + Send + Sync,
    M: Fn(&T) -> U + Send + Sync,
    R: Fn(U, U) -> U + Send + Sync,
{
    data.iter().map(map_fn).fold(init, f: reduce)
}
```

Solution: Map-Reduce

```rust
pub fn map_red<T, U, M, R>(data: &[T], map_fn: M, reduce: R, init: U) -> U
where ···
{
    //data.iter().map(map_fn).fold(init, reduce)
    let chunk_size: usize = data.len() / 2;
    let process_chunk: impl Fn(&[T], U) -> U =
        |input: &[T], init: U| -> U { input.iter().map(&map_fn).fold(init, f: &reduce) };
    std::thread::scope(|s: &Scope| {
        let init_copy: U = init.clone();
        let handle_1: ScopedJoinHandle<U> = s.spawn(|| process_chunk(input: &data[0..chunk_size], init_copy));
        let handle_2: ScopedJoinHandle<U> = s.spawn(|| process_chunk(input: &data[chunk_size..], init));
        let result_1: U = handle_1.join().unwrap();
        let result_2: U = handle_2.join().unwrap();
        reduce(result_1, result_2)
    })
}
```

Questions?

Contact

- @birchmd (Telegram)
- https://github.com/birchmd/
- https://www.typedriven.ca/news/
  - Occasional blog about Rust / blockchain stuff